Diagnostic Tools for Identifying High-Risk Software Modules: A Case-Based Reasoning Approach

Prepared for

NASA Independent Verification and Validation Facility

FAU Technical Report TR-CSE-00-20

Taghi M. Khoshgoftaar*
Edward B. Allen
Yevgeniy Berkovich
Fletcher D. Ross
Florida Atlantic University
Boca Raton, Florida USA

June 2000

*Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel/.

Executive Summary

Various classification modeling techniques have been applied to software quality data. Typically, software metrics are used as independent variables to predict a quality metric, such as whether or not a module is fault-prone. Our prior applied research, sponsored by NASA IV&V Facility, has summarized methods for modeling of fault-prone software modules.

Khoshgoftaar and Allen, and others have observed that published modeling methods may not always produce models of sufficient accuracy. In other words, a modeling technique may not be as robust as it appears in a published study. Thus, more accurate and robust modeling methods are needed.

Case-based reasoning (CBR) is an alternative modeling method based on automated reasoning processes. It has proven useful in a wide variety of domains. CBR is especially useful when there is limited domain knowledge and when an optimal solution process is not known. However, to our knowledge, few CBR systems for software quality modeling have been developed.

A CBR system finds a solution to a new problem based on past experience, represented by cases in a case library. Each case is indexed for quick retrieval according to the problem domain. A solution process algorithm uses a similarity function to measure the relationship between the new problem and each case. The algorithm retrieves relevant cases and determines a solution to the new problem. A CBR system can function as a software quality classification model. The objective is to assign a module to the correct class early in development, i.e., whether it is fault-prone or not. A good "solution" is a class assignment that turns out to be correct after fault data is known.

This paper presents an empirical case study of a very large telecommunications system to illustrate the potential value of CBR as a tool for classifying software modules according to the risk of faults. Our modeling experiments found that a CBR system with raw software product metrics as independent variables was robust compared to selected subsets of metrics, transformation of metrics with principal components analysis, and an expanded set of independent variables with software process metrics. We also explored an innovative classification rule in the CBR context. We compared the CBR models to tree-based models built in prior studies.

CBR has several inherent advantages over other classification techniques for software quality modeling.

- CBR systems can be designed to alert users when a new case is outside the bounds of current experience. This is attractive when an answer of "I don't know" is better than a guess.
- Many CBR systems add or delete cases as new information becomes available, thereby adapting to a changing world.
- CBR is scalable to very large case libraries, and is amenable to concurrent retrieval techniques.

- Once the most similar case(s) has been selected from the library, its detailed description, including qualitative attributes, can help one interpret the automated classification.
- Users of CBR systems can easily see that the solution was derived in a reasonable way, and hence, the CBR system lends itself to user acceptance. CBR systems are not "black boxes".

Keywords: operational software risk, software reliability, faults, fault-prone modules, software metrics, classification, case-based reasoning, CBR

1 Introduction

A recent special issue of *IEEE Software* on software measurement provides a snapshot of the state of the art in the measurement and modeling field [42]. Measurements and models are the means for understanding, controlling, and improving development processes [42]. Successful measurement programs in development organizations are inextricably linked to empirical models that are clearly related to business goals [41]. Our prior applied research, sponsored by NASA IV&V Facility, has summarized methods for modeling of fault-prone software modules [16].

Software product metrics are quantitative attributes of software abstractions. Commonly measured abstractions include call graphs, control flow graphs, and statements. For example, fan-in and fan-out [39] are attributes of a node in a call graph, where each node is an abstraction of a module and each edge represents a call from one to another. Many software product metrics are attributes of a control flow graph in which the nodes represent decision statements or branch destinations and the edges represent potential flow of control. McCabe's cyclomatic complexity is one of the best known in this category [38]. Lines of code is the best known statement metric. Other examples are counts of operators and operands [8]. Commercially available software code analyzers measure more than 50 software product metrics. Our laboratory has the Logiscope and DATRIX packages, as well as a metric analyzer for C which we implemented as a research tool. Logiscope was contributed by Verilog, S.A., and DATRIX was contributed by Bell Canada. Our industry collaborators also have several commercial software product metric analyzers.

Software process metrics can be derived from problem reporting systems and configu-

ration management systems. Process metrics are especially important for legacy systems and developments with significant reuse. Research by Khoshgoftaar et al. found that reuse [25, 26], the history of corrected faults [21], and the experience of programmers [22] can be significantly associated with faults.

Various classification modeling techniques have been applied to software quality data. Our research group has used discriminant analysis [25], logistic regression [12], decision trees [18, 43], and artificial neural networks [32, 33] for software quality modeling. In addition, others have used discriminant power [44], optimal set reduction [2], and fuzzy classification [5]. In this paper, a *Type I* misclassification is when a model classifies a software module as fault-prone which is actually not fault-prone, and a *Type II* misclassification is when a model classifies a software module as not fault-prone which is actually fault-prone. A *misclassification rate* is the number misclassified divided by the actual number in the class.

Khoshgoftaar and Allen [17], and others [36] have observed that published modeling methods may not always produce models of sufficient accuracy. In other words, a modeling technique may not be as robust as it appears in a published study. Thus, more accurate and robust modeling methods are needed.

Case-based reasoning (CBR) is an alternative modeling method based on automated reasoning processes. It has proven useful in a wide variety of domains [35] including software cost estimation, software reuse, software design, and software help desk. CBR is especially useful when there is limited domain knowledge and when an optimal solution process is not known [37]. A recent book [37] presents the state of the art in CBR, the lessons learned from specific applications, and directions for the future. However, to our knowledge, few CBR systems for software quality modeling have been developed [7, 30].

A CBR *system* is a software system that implements specific data structures, algorithms, and policies as *design features*. A CBR system finds a solution to a new problem based on past experience, represented by *cases* in a *case library*. Each case is *indexed* for quick retrieval according to the problem domain. A *solution process* algorithm uses a *similarity function* to measure the relationship between the new problem and each case. The algorithm retrieves relevant cases and determines a solution to the new problem.

A CBR system can function as a software quality classification model. The "problem" is to assign a module to the correct class early in development, i.e., whether it is fault-prone or not. A good "solution" is a class assignment that turns out to be correct after fault data is known. A *case* consists of all available information on a module. This includes whether it is fault-prone or not, its product attributes, and its process history. Software product metrics are *indices* representing module attributes. Process metrics are *indices* representing the process history.

In principle, CBR has several advantages over other classification techniques for software quality modeling.

- CBR systems can be designed to alert users when a new case is outside the bounds of current experience. This is attractive when an answer of "I don't know" is better than a guess. In contrast, a typical classification model always gives some kind of answer, even in extreme cases.

- Many CBR systems add or delete cases as new information becomes available, thereby adapting to a changing world [48]. Other techniques, on the other hand, may require frequent reestimation of model parameters to track recent data. We suspect that retrieval and decision rules are usually more stable than model pa-

rameters of other techniques.

- CBR is scalable to very large case libraries, and is amenable to concurrent retrieval techniques [11]. This means that fast retrieval is feasible as the size of the case library scales up.

- Once the most similar case(s) has been selected from the library, its detailed description, including qualitative attributes, can help one interpret the automated classification. Most other classification techniques, on the other hand, provide little concrete interpretive information.

- Users of CBR systems can easily see that the solution was derived in a reasonable way, and hence, the CBR system lends itself to user acceptance. CBR systems are not "black boxes".

This paper presents an empirical case study of a very large telecommunications system to illustrate the potential value of CBR as a tool for classifying faulty software modules.

The remainder of this paper gives an overview of CBR, empirical results, and conclusions. The empirical case study used the Software Measurement Analysis and Reliability Toolkit (SMART) [19], which was developed in 1998 with support from the Empirical Software Engineering Laboratory at Florida Atlantic University, and industrial partners. Appendix A summarizes features of SMART; Appendix B explains calculations for principal components analysis; and Appendix C presents detailed results of experiments.

Table 1: Notation

Symbol	Definition
i	Index for unclassified modules
j	Index for cases, $j = 1, \ldots, n$
k	Index for metrics, $k = 1, \ldots, m$
x_{ik}	Value of metric k of module i
$\mathbf{x}_i$	Vector of metrics values for module i
c_{jk}	Value of metric k of case j
$\mathbf{c}_j$	Vector of metric values for case j
y_j	Number of faults in module j
w_k	Weights
α	Scale factor
d_{ij}	Similarity (i.e., distance) between $\mathbf{x}_i$ and $\mathbf{c}_j$
d_{fp}	Average distance to fault-prone nearest neighbors
d_{nfp}	Average distance to not fault-prone nearest neighbors
$\mathcal{N}$	Set of nearest-neighbor cases
$n_{\mathcal{N}}$	Number of cases in $\mathcal{N}$
$Class(\mathbf{x}_i)$	Predicted class of module i
C_I	Cost of a Type I misclassification
C_{II}	Cost of a Type II misclassification

2 Case-Based Reasoning

Case-based reasoning finds solutions to new problems based on past experience, represented by cases in a case library. The case library and the associated retrieval and decision rules constitute a CBR model. For an introduction to case-based reasoning, see [34]. Table 1 lists notation introduced in this section.

In this paper, we focus on classification problems in software quality modeling. Suppose each case in the library has known attributes and class membership. Given a case with unknown class, we predict its class to be the same as the class of the most similar case(s) in the library, where similarity is defined in terms of case attributes. This paper applies this approach to classification of software modules as fault-prone, or not.

Case-based reasoning is often compared to rule-based inference and model-based problem solving systems [34]. Application to software engineering problem solving is a topic for further research [47].

A case consists of all available information on a module. This includes its class and all measurements. In this study, the case library consists of the *fit* data set, having n modules. The case library represents the past experiences of the development organization. Let $\mathbf{c}_j$ be the vector of attributes of the j^{th} module in the case library, and let c_{jk} be the k^{th} attribute of that module.

SMART offer the option to standardize all variables, using the following equation.

$$z_{ik} = \frac{x_{ik} - \bar{x}_k}{s_k} \tag{1}$$

where z_{ik} is the standardized value and $\bar{x}_k$ and s_k are the mean and standard deviation over i for all x_{ik}, respectively. This option also performs a similar transformation all the cases, c_{jk}. In the following, the notation x_{ik} and c_{jk} indicate either raw or standardized values according to the context.

Similarity function. From these past cases we want to extract a few that most closely resembles our unclassified module. Let $\mathbf{x}_i$ be the vector of attributes of the i^{th} unclassified module, and x_{ik} be the k^{th} attribute of that module. Several measures of similarity are presented in the literature according to the problem domain, the availability of attribute data, and whether data types are categorical, discrete, real, etc. [34]. Since our data is strictly quantitative, we think of "similarity" as the "distance" between cases.

Euclidean Distance considers each independent variable as a dimension of an m-dimensional space. A module is represented by a point in this space. The Euclidean

distance between a module and a case is their weighted distance in this space. The weights, if any, are those provided by the analyst.

$$d_{ij} = \left(\sum_{k=1}^{m} (w_k(c_{jk} - x_{ik}))^2 \right)^{1/2} \quad (2)$$

Nearest neighbors. After the tool calculates distances between a module $\mathbf{x}_i$ and all cases using the similarity function, the distances are sorted. The cases, $\mathbf{c}_j$, with the smallest distances, d_{ij}, are of primary interest. The set of nearest neighbors, $\mathcal{N}$, is an input to the solution algorithm below. The analyst selects the number of nearest neighbors. Based on a preliminary empirical investigation with software quality data, $n_{\mathcal{N}} \in \{1, 3, 5, 7, 9\}$ appears to be adequate [19].

Solution algorithm. Each solution algorithm assigns the unclassified module to a class. Predictions of a quantitative dependent variable made by the solution algorithm can be scaled in order to improve the accuracy of class predictions. Many solution algorithms supported by SMART calculate $\hat{y}_i$ and then classify the module by

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{If } \alpha\hat{y}_i < \theta \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \quad (3)$$

In our case studies, we use $\theta = 1$. To choose a preferred value of α, the analyst designates the fit data set as the case library and also as the target data set, and supplies a list of candidate scale factors, α. The analyst can then choose a preferred value of α based on experiment results calculated by cross-validation. When the target is the test data set or when the target is a current data set, the analyst can then specify the preferred scale factor, α.

The *unweighted average* solution algorithm averages the dependent variable, y_j, of the closest $n_{\mathcal{N}}$ modules from the case library to form a value of $\hat{y}_i$ for the target module.

$$\hat{y}_i = \frac{1}{n_{\mathcal{N}}} \sum_{j \in \mathcal{N}} y_j \tag{4}$$

In this study, y_j is the number of faults discovered by customers. We are primarily concerned with classification, so the value predicted is not as important as the predicted class, given by Equation (3).

Data clustering. In addition to the basic CBR model, the case study examined an extension by cluster analysis to enhance classification. The case library is partitioned into clusters according to the actual class of each case. SMART compares a module to the fault-prone cluster and the not fault-prone cluster and determines the closest group. SMART supports the same similarity functions and solution algorithms here as in the classic CBR model.

Instead of scale factors, this approach provides for a list of cost ratios, C_I/C_{II}. The analyst can calculate a cost ratio as the ratio of the cost of a Type I misclassification, C_I, to the cost of a Type II misclassification, C_{II}. However, this is often difficult to estimate, and therefore, SMART provides for experiments with a list of values. In software quality modeling, a Type II misclassification can be much more serious than a Type I, because the cost of releasing fault-prone modules to customers is usually much more expensive than wasting effort enhancing low-risk modules. The analyst can experiment to choose a preferred C_I/C_{II} value.

The classification rule used here is based on our recent work with statistical classification techniques [23]. For an unclassified module, $\mathbf{x}_i$, let $d_{nfp}(\mathbf{x}_i)$ be the average distance

to not fault-prone nearest-neighbor cases, and let $d_{fp}(\mathbf{x}_i)$ be the average distance to fault-prone nearest-neighbor cases. The module is classified by

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{If } \frac{d_{fp}(\mathbf{x}_i)}{d_{nfp}(\mathbf{x}_i)} > \frac{C_I}{C_{II}} \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \quad (5)$$

where C_I/C_{II} is experimentally chosen. The analyst can choose C_I/C_{II} by a similar method as α, described above.

3 Empirical Case Study

This section presents Berkovich's empirical investigation of the advantages of CBR for software quality modeling [1].

3.1 System Description

We studied the same system as [24] and [29]. We conducted a case study of a very large legacy telecommunications system, written in a high-level language (Protel) using the procedural development paradigm, and maintained by professional programmers in a large organization. The entire system had significantly more than ten million lines of code. This embedded-computer application included numerous finite-state machines. We studied four consecutive releases, which we label 1 through 4 in this paper. In this study, the *fit* data set consisted of measurements of Release 1, and *test* data sets consisted of measurements of Releases 2, 3, and 4. Even though the software was significantly enhanced from release to release, the project staff considered the software development process to be stable.

A *module* consisted of a set of related source-code files. Fault data was collected at the module-level by the problem reporting system. A module was considered fault-prone if any faults discovered by customers resulted in changes to source code in the module, and not fault-prone otherwise. Faults discovered in deployed telecommunications systems are typically extremely expensive, because, in addition to down-time due to failures, visits to customer sites are usually required to repair them.

Analysis of configuration-management data identified modules that were unchanged from the prior release. More than 99% of the unchanged modules had no faults. There were too few fault-prone modules for effective modeling. This case study considered only updated modules, that is, those that were new or had at least one update to source code since the prior release. For modeling, we selected updated modules with no missing data in relevant variables. These modules had several million lines of code in a few thousand modules in each release.

Fault data was collected from the problem reporting system. Problem reports were tabulated and anomalies were resolved. Table 2 summarizes the distribution of faults discovered by customers. The proportion of modules with no faults among the updated modules of the *fit* data set (Release 1) was 0.937, and the proportion with at least one fault was 0.063. Such a small set of modules is often difficult to identify early in development. In this study, due to a lack of detailed data, we assumed that customers used the releases a similar amount of time. However, we suspect that each release was used somewhat less than its predecessors. Comparison of fault-discovery rates across releases is a topic for future research.

Pragmatic considerations usually determine the set of available software metrics. We do not advocate collecting a particular set of metrics for software quality models to the

Table 2: Distribution of faults discovered by customers [24]

Percentage of updated modules				
	Release			
Faults	1	2	3	4
0	93.7%	95.3%	98.7%	97.7%
1	5.1%	3.9%	1.0%	2.1%
2	0.7%	0.7%	0.2%	0.2%
3	0.3%	0.1%	0.1%	0.1%
4	0.1%	*		
6	*			
9	*			

* one module

exclusion of others recommended in the literature. We prefer a data-mining approach to exploiting metric data [6], analyzing a broad set of metrics, rather than limiting data collection according to predetermined research questions. A small set of predetermined measures may not be significant in all development environments due to variation in development processes. Thus, we prefer to analyze a large set of candidate software metrics.

This project used Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD), which is a decision-support system that includes software-measurement facilities and software quality models [9]. Because marginal data collection costs were modest, EMERALD provided over fifty source-code metrics. Preliminary data analysis selected product metrics aggregated to the module level that were appropriate for modeling purposes, as listed in Table 3. Counts of procedure calls were derived from call graphs. Some product metrics were measures of a module's control flow graph, which consists of nodes and arcs depicting the flow of control. Other product metrics quantified attributes of statements.

Process metrics listed in Table 4 were tabulated from the configuration management system which maintained records regarding updates by each designer, and from the problem reporting system which maintained records on past problems.

Execution metrics listed in Table 5 were forecast from deployment records [10] and laboratory measurements of an earlier release. Future research will refine these metrics.

EMERALD helps software designers and managers to assess risks of embedded software and thereby to improve software quality [9]. It was developed by Nortel Networks (formerly Northern Telecom) in partnership with Bell Canada and others. At various points in the development process, EMERALD's software quality models predict module risk based on available measurements.

3.2 Preprocessing Data

For models using raw metrics as independent variables, we transformed all variables using Equation (1) so that they would all have the same unit of measure, namely, a standard deviation.

For models involving principal components, this empirical study preprocessed data in the same manner as [46]. Preliminary data analysis [14] found that the execution metrics were each not correlated very much with any other metrics. Moreover, the product metrics were not correlated with the process metrics. However, the product metrics were highly correlated with each other and similarly, the process metrics were highly correlated with each other. For this study, we transformed the 24 product metrics listed above with principal components analysis (PCA) [45], resulting in 6 principal components ("domain metrics"). Details on principal components analysis are in Appendix B [45]. The domain

Table 3: Software product metrics [29]

Symbol	Description
Call Graph Metrics	
CALUNQ	Number of distinct procedure calls to others.
CAL2	Number of second and following calls to others. $CAL2 = CAL - CALUNQ$ where *CAL* is the total number of calls.
Control Flow Graph Metrics	
CNDNOT	Number of arcs that are not conditional arcs.
IFTH	Number of non-loop conditional arcs, i.e., if-then constructs.
LOP	Number of loop constructs.
CNDSPNSM	Total span of branches of conditional arcs. The unit of measure is arcs.
CNDSPNMX	Maximum span of branches of conditional arcs.
CTRNSTMX	Maximum control structure nesting.
KNT	Number of knots. A "knot" in a control flow graph is where arcs cross due to a violation of structured programming principles.
NDSINT	Number of internal nodes (i.e., not an entry, exit, or pending node).
NDSENT	Number of entry nodes.
NDSEXT	Number of exit nodes.
NDSPND	Number of pending nodes, i.e., dead code segments.
LGPATH	Base 2 logarithm of the number of independent paths.
Statement Metrics	
FILINCUQ	Number of distinct include files.
LOC	Number of lines of code.
STMCTL	Number of control statements.
STMDEC	Number of declarative statements.
STMEXE	Number of executable statements.
VARGLBUS	Number of global variables used.
VARSPNSM	Total span of variables.
VARSPNMX	Maximum span of variables.
VARUSDUQ	Number of distinct variables used.
VARUSD2	Number of second and following uses of variables. $VARUSD2 = VARUSD - VARUSDUQ$ where *VARUSD* is the total number of variable uses.

Table 4: Software process metrics [29]

Symbol	Description
DES_PR	Number of problems found by designers during development of the current release.
BETA_PR	Number of problems found during beta testing of the current release.
DES_FIX	Number of problems fixed that were found by designers in the prior release.
BETA_FIX	Number of problems fixed that were found by beta testing of the prior release.
CUST_FIX	Number of problems fixed that were found by customers in the prior release.
REQ_UPD	Number of changes to the code due to new requirements.
TOT_UPD	Total number of changes to the code for any reason.
REQ	Number of distinct requirements that caused changes to the module.
SRC_GRO	Net increase in lines of code.
SRC_MOD	Net new and changed lines of code (deleted lines are not counted).
UNQ_DES	Number of different designers making changes.
VLO_UPD	Number of updates to this module by designers who had 10 or less total updates in entire company career.
LO_UPD	Number of updates to this module by designers who had between 11 and 20 total updates in entire company career.
UPD_CAR	Number of updates that designers had in their company careers.

Table 5: Software execution metrics [29]

Symbol	Description
USAGE	Deployment percentage of the module.
RESCPU	Execution time of an average transaction on a system serving consumers.
BUSCPU	Execution time of an average transaction on a system serving businesses.
TANCPU	Execution time of an average transaction on a tandem system.

Table 6: Factor pattern for principal components of product metrics.

	*Prod*1	*Prod*2	*Prod*3	*Prod*4	*Prod*5	*Prod*6
CALUNQ	**0.90241**	0.05180	0.10442	0.23226	0.17394	0.06161
VARUSDUQ	**0.89496**	0.18889	0.15268	0.17704	0.14681	0.19375
LOC	**0.88610**	0.28067	0.18160	0.16929	0.16431	0.14445
NDSENT	**0.87966**	-0.11142	0.01770	0.18394	0.10988	0.17201
STMEXE	**0.86869**	0.25870	0.17612	0.17324	0.26880	0.07169
STMCTL	**0.86701**	0.26070	0.27411	0.17258	0.08509	0.17429
NDSEXT	**0.84668**	0.01970	0.10855	0.20099	0.08568	0.35294
STMDEC	**0.84595**	0.20127	0.14148	0.14922	0.07117	0.14898
IFTH	**0.84569**	0.34158	0.27880	0.18162	0.10404	0.10659
NDSINT	**0.84185**	0.34355	0.27606	0.15248	0.18487	0.10920
CNDNOT	**0.83478**	0.31173	0.26233	0.15217	0.23697	0.17495
LOP	**0.82816**	0.10817	0.20842	0.01714	0.02129	-0.09590
VARGLBUS	**0.80191**	0.35962	0.20123	0.14369	0.21197	0.20453
VARUSD2	**0.79088**	0.44096	0.27108	0.11186	0.18082	0.12928
CAL2	**0.59715**	0.20418	0.07284	0.19317	0.56903	-0.05255
VARSPNSM	0.39174	**0.86022**	0.17718	0.10430	0.06747	0.08423
VARSPNMX	0.14039	**0.83489**	0.17722	0.35150	0.10357	0.09136
CNDSPNMX	0.12121	0.27629	**0.75661**	0.14289	0.25648	0.30600
CTRNSTMX	0.32233	0.09595	**0.70922**	0.42101	-0.00726	-0.01574
CNDSPNSM	0.60974	0.21553	**0.64240**	0.00704	0.22007	0.13087
FILINCUQ	0.39561	0.25790	0.15541	**0.72651**	-0.03570	0.16963
LGPATH	0.21017	0.37957	0.35793	**0.63962**	0.16986	-0.04151
KNT	0.21362	0.06906	0.17464	-0.00640	**0.88896**	0.09719
NDSPND	0.40212	0.14886	0.21690	0.07507	0.08412	**0.81557**
Variance	11.61638	2.82091	2.37167	1.69515	1.64281	1.23002
% Var.	48.40%	11.75%	9.88%	7.06%	6.85%	5.13%
Cum. %	48.40%	60.15%	70.03%	77.09%	83.94%	89.07%

Stopping rule: at least 89% of variance

metrics are named *Prod*1 through *Prod*6. Table 6 shows the rotated factor pattern. Each table entry is the correlation between a product metric and a domain metric. To aid interpretation, the largest value in each row is bold

We also transformed the 14 process metrics listed above with principal components analysis, resulting in 8 principal components ("domain metrics"). The domain metrics

are named *Prcs*1 through *Prcs*8. Table 7 shows the rotated factor pattern.

Not only is principal components analysis a technique for transforming data into orthogonal variables, it also is a data reduction technique which ignores insignificant variation in the data. As shown in Table 6 and 7, both principal components analyses accounted for more than 89% of the variance. The 24 product metrics were reduced to only 6 domain metrics, and the 14 process metrics were reduced to only 8 domain metrics.

3.3 Empirical Results

For ease of reference, we used descriptive names for the models used in the research. Table 8 summarizes the independent variables of each model in this case study. Table 9 summarizes the accuracy of all these models. Detailed results are presented in Appendix C [1].

Model CART-28 [40] had twenty-eight candidate independent variables, as shown in Table 8, of which CART found four to be significant. Model RAW-4 was a CBR model based on the same four independent variables, standardized. Similarly, model CART-42 [40] had forty-two candidates of which eleven were significant. Model RAW-11 was a CBR model based on the same eleven independent variables, standardized.[1] We denote standardized variables by prime ($'$). Specifically, model RAW-4 employed *USAGE*$'$, *FILINCUQ*$'$, *NDSENT*$'$, and *VARSPNMX*$'$ variables. Model RAW-11 used *TOT_UPD*$'$, *SRC_MOD*$'$, *USAGE*$'$, *UNQ_DES*$'$, *UPD_CAR*$'$, *CNDSPNSM*$'$, *FILINCUQ*$'$, *NDSENT*$'$, *NDSINT*$'$, *STMCTL*$'$, and *VARSPNMX*$'$ metrics.

Model CART-10 [46] had ten candidate independent variables, as shown in Table 8,

[1] Note that standardization is irrelevant to a tree-based model, but is relevant to calculating similarity.

Table 7: Factor pattern for principal components of process metrics.

	*Prcs*1	*Prcs*2	*Prcs*3	*Prcs*4	*Prcs*5	*Prcs*6	*Prcs*7	*Prcs*8
REQ_UPD	**0.90643**	0.18040	0.17360	-0.01026	0.01492	0.05107	0.00541	0.11665
REQ	**0.81854**	0.28570	-0.03743	-0.20836	-0.00660	0.06722	0.08157	0.08421
TOT_UPD	**0.79828**	0.29250	0.18792	0.29906	0.14177	0.09401	0.06932	0.24259
UPD_CAR	**0.76799**	-0.18604	0.11188	0.46925	0.05774	0.05102	-0.04216	-0.11040
UNQ_DES	**0.75105**	0.35609	0.02749	0.25565	0.19213	0.13293	0.14578	0.21690
VLO_UPD	0.18565	**0.91828**	0.05975	0.08848	0.05386	0.05235	0.02823	-0.00220
LO_UPD	0.30638	**0.85506**	0.11794	0.12719	0.05238	0.02050	0.06262	0.19528
SRC_GRO	0.04399	0.07004	**0.93638**	0.00451	0.01839	-0.02000	0.02660	-0.00832
SRC_MOD	0.17405	0.07582	**0.90246**	0.07652	0.00979	0.03290	0.00115	0.17766
DES_FIX	0.12698	0.22955	0.04702	**0.86977**	0.03807	0.06639	0.04361	0.27041
CUST_FIX	0.12043	0.08325	0.02270	0.04039	**0.97240**	0.15339	0.03259	0.06120
BETA_PR	0.14929	0.05732	0.00376	0.06531	0.15792	**0.96461**	0.02145	0.11591
BETA_FIX	0.08989	0.06419	0.02413	0.02982	0.03149	0.01964	**0.98936**	0.00262
DES_PR	0.27525	0.15597	0.19269	0.29835	0.08462	0.16218	-0.00160	**0.83048**
Variance	3.58010	2.03751	1.82929	1.30069	1.04678	1.02598	1.02220	0.98856
% Var.	25.57%	14.55%	13.07%	9.29%	7.47%	7.33%	7.30%	7.06%
Cum. %	25.57%	40.12%	53.19%	62.48%	69.95%	77.28%	84.58%	91.64%

Stopping rule: at least 89% of variance

Table 8: Models

	Independent Variables Software Metrics		
Model	Product	Process	Execution
CART-28 [40]	24 raw		4 raw
CART-42 [40]	24 raw	14 raw	4 raw
CART-10 [46]	6 PCA		4 raw
CART-18 [46]	6 PCA	8 PCA	4 raw
RAW-4	4 standardized selected by CART-28		
RAW-11	11 standardized selected by CART-42		
RAW-28	24 standardized		4 standardized
RAW-42	24 standardized	14 standardized	4 standardized
PCA-5	5 selected by CART-10		
PCA-7	7 selected by CART-18		
PCA-10	6 PCA		4 raw
PCA-18	6 PCA	8 PCA	4 raw

Table 9: Summary of accuracy of models

	Misclassification rates							
	Release 1		Release 2		Release 3		Release 4	
Model	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
CART-28 [40]	27.6%	29.7%	24.4%	29.6%	25.5%	25.5%	30.3%	27.2%
CART-42 [40]	27.7%	26.6%	27.9%	28.6%	30.4%	34.0%	33.7%	27.2%
CART-10 [46]	30.4%	30.6%	26.6%	24.9%	28.8%	21.3%	32.7%	27.2%
CART-18 [46]	38.7%	22.3%	29.3%	21.2%	29.9%	19.1%	32.7%	19.6%
RAW-4	30.3%	27.5%	30.6%	24.9%	32.2%	21.3%	39.6%	29.3%
RAW-11	28.9%	30.1%	31.5%	24.9%	32.7%	21.3%	41.0%	34.0%
RAW-28	30.4%	31.4%	30.5%	31.2%	32.5%	25.5%	42.4%	22.9%
RAW-42	27.7%	31.0%	28.1%	32.3%	29.6%	25.5%	39.4%	30.4%
PCA-5	31.1%	35.8%	31.8%	28.6%	33.8%	29.8%	41.5%	28.3%
PCA-7	34.3%	32.6%	28.5%	38.1%	29.7%	21.3%	34.3%	32.6%
PCA-10	32.1%	28.6%	32.1%	28.8%	33.9%	27.7%	38.1%	27.2%
PCA-18	25.6%	35.4%	26.7%	35.0%	27.4%	27.7%	36.1%	29.3%

The numbers are rounded to the nearest tenth.

Using scale factor approach.

of which CART found five to be significant. Model PCA-5 was a CBR model based on the same five independent variables. Similarly, model CART-18 [46] had eighteen candidates of which seven were significant. Model PCA-7 was a CBR model based on the same seven independent variables. Specifically, model PCA-5 used *USAGE*, *TANCPU*, *Prod1*, *Prod2*, and *Prod4* principal components variables. And, finally, model PCA-7 employed *USAGE*, *Prcs1*, *Prcs5*, *Prcs6*, *Prod1*, *Prod2*, and *Prod4* principal components metrics.

Uniformly weighted raw metrics as predictors. We ran the SMART tool against the entire set of 42 standardized metrics available for our case study. Here, we used equal weights for all the metrics and Euclidean distance as a distance algorithm. We performed our experiments for a single case, and for the clusters of 3, 5, 7, and 9 cases. We wanted to find out how to configure SMART's parameters to allow us to achieve optimal results. For this experiment we used the data sets including modules from Releases 1 and 2 with all 42 standardized metrics.

Our hypothesis was that we will achieve better results, in other words, build a model that performs better, by using the 7 or 9 most similar modules from the case library for each module in test data set.

As we are able to see in Table 10, the best results (shown bold) were achieved when we build models with 7 or 9 most similar modules (i.e., cases) as opposed to models with 1, 3, or 5 similar cases. These results clearly support our hypothesis. Restricting the CBR system to only one nearest neighbor is vulnerable to misleading results, because the one chosen may be an outlier. When the set of nearest neighbors has 7 or 9 cases, the effect of an outlier, if any, is diluted.

Table 10: Uniform weights for model RAW-42

	Release 1		Release 2	
Neighbors	Type I	Type II	Type I	Type II
1 case	4.883%	81.223%	5.169%	79.365%
3 cases	12.895%	57.205%	13.423%	62.434%
5 cases	18.918%	46.288%	18.486%	47.619%
7 cases	**23.947%**	**35.371%**	**23.470%**	**39.153%**
9 cases	**27.749%**	**31.004%**	**28.085%**	**32.275%**

Weighting USAGE. We conducted a set of experiments with different weights for the *USAGE* metric.[2] (See Equation (2).) All other metrics had a weight of one. Prior research had found *USAGE* to be an important variable [10]. Our intuition suggested that a larger weight for *USAGE*, which would mean the increased contribution of *USAGE* to the model, should produce a better model.

Our hypothesis was that in most cases, the models with a higher weight of *USAGE* will perform much better than those using a lower weight of *USAGE*.

The tables in Appendix C show that for most experiments the best models use the weight *USAGE* higher than 1. In fact, in most of the models, we preferred *USAGE* with a weight between 5 and 10. As one can observe in the tables in Appendix C, in all cases, the preferred weight of *USAGE* turned out to be a significant factor in quality prediction.

Experiments with the scale factor. The scale factor, α, in Equation (3) is a very important parameter in building a good model. Scale factors, when applied to the model can give more weight to the modules of a particular type. In this research, we used the scale factors to give more weight to the fault-prone modules.

In this study, the scale factor was empirically chosen by looking at the misclassifica-

[2]Similarly, weights were chosen for *USAGE'*.

tion rates for cross-validation of Release 1 and for predictions of Release 2. We preferred approximately equal Type I and Type II misclassification rates. The Table 11 shows how different the model's accuracy is for different scale factors. The model used for this experiment was the RAW-28 Model that uses all 28 product and execution raw metrics. As indicated in Table 11, we used various scale factors, α, to find out how scale factors affect model RAW-28. As we were able to determine, there was a very significant difference in misclassification rates for different α. For example, with a scale factor $\alpha = 1.0$, the Type I and Type II misclassification rates are extremely unbalanced, for example, 0.409% and 98.253% respectively, for Release 1. But, for $\alpha = 9.0$, the misclassification rates are very balanced, 30.351% and 31.441% respectively, for Release 1.

Principal components as predictors. Based on the experiments presented in Appendix C and summarized in Table 12, we were able to make following observations for the scale factor approach (Equation (3)).

Model PCA-5 had the predictors chosen to be significant by [46]. Model PCA-10 had all six product domain metrics and four raw execution metrics as predictors. Comparison of PCA-10 and PCA-5 showed that they performed very similarly. It was very hard to determine any significant improvement of using PCA-5 versus PCA-10, as can be observed in Table 12. This illustrates that CBR can be a robust tool for software quality prediction and does not need a model selection mechanism in order to successfully make software quality prediction. The metric selection made for PCA-5 did not seem to have any significant positive effect on the models' accuracies.

Model PCA-7 had predictors chosen to be significant by [46]. Model PCA-18 had all 6 product domain metrics, all 8 process domain metrics, and all 4 raw execution metrics

Table 11: Impact of scale factor on accuracy of model RAW-28

	Misclassification rates							
	Release 1		Release 2		Release 3		Release 4	
α	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
1.00	0.409%	98.253%	0.343%	93.651%	0.716%	100.000%	0.926%	89.130%
1.20	0.439%	95.633%	0.422%	93.122%	0.830%	100.000%	1.107%	86.957%
1.40	0.585%	93.886%	0.633%	91.005%	1.030%	97.872%	1.493%	84.783%
1.60	0.965%	92.140%	1.134%	88.360%	1.631%	95.745%	2.213%	80.435%
1.80	1.696%	84.279%	1.688%	83.598%	2.490%	91.489%	3.423%	75.000%
2.00	1.696%	84.279%	1.688%	83.598%	2.490%	91.489%	3.423%	75.000%
2.20	1.696%	84.279%	1.688%	83.598%	2.490%	91.489%	3.423%	75.000%
2.40	3.509%	79.913%	3.112%	76.190%	4.350%	76.596%	6.613%	67.391%
2.60	3.509%	79.913%	3.112%	76.190%	4.350%	76.596%	6.613%	67.391%
2.80	3.509%	79.913%	3.112%	76.190%	4.350%	76.596%	6.613%	67.391%
3.00	6.637%	68.996%	6.250%	68.254%	7.527%	55.319%	11.863%	55.435%
4.00	6.637%	68.996%	6.250%	68.254%	7.527%	55.319%	11.863%	55.435%
5.00	13.684%	54.585%	13.001%	53.439%	14.997%	40.426%	21.951%	39.130%
6.00	13.684%	54.585%	13.001%	53.439%	14.997%	40.426%	21.951%	39.130%
7.00	13.684%	54.585%	13.001%	53.439%	14.997%	40.426%	21.951%	39.130%
8.00	13.684%	54.585%	13.001%	53.439%	14.997%	40.426%	21.951%	39.130%
9.00	**30.351%**	**31.441%**	**30.459%**	**31.217%**	**32.570%**	**25.532%**	**42.383%**	**22.826%**
10.00	30.351%	31.441%	30.459%	31.217%	32.570%	25.532%	42.383%	22.826%
20.00	30.351%	31.441%	30.459%	31.217%	32.570%	25.532%	42.383%	22.826%
50.00	30.351%	31.441%	30.459%	31.217%	32.570%	25.532%	42.383%	22.826%

Table 12: Accuracy of PCA models

	Release 1		Release 2		Release 3		Release 4	
Model	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
PCA-5	31.1%	35.8%	31.8%	28.6%	33.8%	29.8%	41.5%	28.3%
PCA-7	34.3%	32.6%	28.5%	38.1%	29.7%	21.3%	34.3%	32.6%
PCA-10	32.1%	28.6%	32.1%	28.8%	33.9%	27.7%	38.1%	27.2%
PCA-18	25.6%	35.4%	26.7%	35.0%	27.4%	27.7%	36.1%	29.3%

This is a subset of Table 9.
Using scale-factor approach.

as predictors. Comparison between our PCA-18 and PCA-7 showed that there was a very small difference in the models' accuracies with a slight advantage to the PCA-18 Model, as can be seen in Table 12. In this case, as well as in the previous one, we can see that the prior selection of the metrics did not improve significantly the models and their accuracies.

Comparisons of raw metrics and principal components as predictors. Comparing our raw metrics models to other models, we discovered that the raw metric models performed slightly better than all the other models, which were using the PCA-generated metrics and the scale-factor approach. This can be seen by inspecting Tables 19 through 24 and 37 through 39 from Appendix C for raw metric models, and Tables 25 through 36 for PCA models, which are summarized in Table 9. The difference was not very large — only about 1 to 3 percentage points on the average. This comparison seems to imply that SMART-generated models' accuracies using the scale-factor approach does not improve significantly if standardized raw metrics are replaced with PCA-generated metrics.

Table 13: Accuracy of RAW models

	Release 1		Release 2		Release 3		Release 4	
Model	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
RAW-4	30.3%	27.5%	30.6%	24.9%	32.2%	21.3%	39.6%	29.3%
RAW-11	28.9%	30.1%	31.5%	24.9%	32.7%	21.3%	41.0%	34.0%

This is a subset of Table 9.

Process metrics as additional predictors. We wanted to find out whether the models that use product and execution metrics as well as process metrics will have lower misclassification rates than those that only use product and execution metrics. Of all of our models, RAW-4, RAW-28, PCA-5, and PCA-10 were using only product and execution metrics, while the models RAW-11, PCA-7, and PCA-18 were using product, process, and execution metrics.

For the purposes of our experiments using the scale-factor approach, we wanted to examine the following three comparisons:

- RAW-4 vs. RAW-11
- PCA-5 vs. PCA-7
- PCA-10 vs. PCA-18

For the first comparison we used the data presented in Appendix C from Tables 19, 20, and 21 to compare against the results displayed in Tables 22, 23, and 24. As can be seen from Table 13, the models performed almost identically. This allows us to suggest that adding the process metrics into these scale-factor CBR models did not bring any improvement.

For the second comparison, we used data presented in Appendix C in Tables 28, 29, and 30 and compared it against the data displayed in Tables 25, 26, and 27. After studying this data, as well as Table 12, we came to a conclusion, that the model using the process metrics, PCA-7, did, indeed, had a slightly better accuracy than the PCA-5 model. But this advantage was very inconclusive and could not be considered significant.

And, lastly, we compared the results presented in Appendix C from Tables 34, 35, and 36 against the results displayed in Tables 31, 32, and 33. After inspecting these tables, as well as Table 12, we concluded that, once again, use of the process metrics did not create any improvement over the scale-factor CBR model that did not use the process metrics.

Experiments with cost ratios. We carried out a set of experiments to find out whether we could produce better models using various cost-ratios, C_I/C_{II}, than the models we obtained using the scale factor, α. We ran these experiments for our PCA-generated models (PCA-5, PCA-7, PCA-10, and PCA-18). In Table 14, we can see the best results for each of the PCA-generated models.

As can be seen from the Table 17, the models created using the cost-ratio (C_I/C_{II}) approach are similar or better than the models built with scale factor (α) approach. In fact, for the PCA-10 model, the accuracy using scale factor approach was significantly better, while, for the three other models, the accuracy was somewhat better using cost ratio approach. It seems that cost-ratio approach takes fuller advantage of the models that use only the most significant PCA domain metrics, PCA-5 and PCA-7. In order to select the best cost-ratio, we experimented with various values of cost ratio, C_I/C_{II}. Based on the cross-validation results from Release 1, we were able to select the cost-ratio

Table 14: Experiments Using Cost Ratios

Model	Release	weight of *USAGE*	Neighbors	C_I/C_{II}	Misclassifications Type I	Type II
PCA-10	1	9	9	0.59	30.468%	31.441%
PCA-10	2	9	9	0.59	23.101%	41.270%
PCA-10	3	9	9	0.59	28.678%	36.170%
PCA-10	4	9	9	0.59	39.012%	26.087%
PCA-5	1	17	9	0.46	26.257%	27.074%
PCA-5	2	17	9	0.46	22.310%	32.804%
PCA-5	3	17	9	0.46	25.644%	29.787%
PCA-5	4	17	9	0.46	37.983%	28.261%
PCA-18	1	9	9	0.65	28.947%	27.511%
PCA-18	2	9	9	0.65	29.088%	26.984%
PCA-18	3	9	9	0.65	37.979%	21.277%
PCA-18	4	9	9	0.65	41.765%	19.149%
PCA-7	1	10	9	0.54	25.000%	24.891%
PCA-7	2	10	9	0.54	24.815%	29.101%
PCA-7	3	10	9	0.54	27.418%	19.149%
PCA-7	4	10	9	0.54	29.671%	28.261%

with the most balanced misclassification rates. (Release 2 results were not used when choosing C_I/C_{II}.)

Tables 15 and 16 illustrate the importance of using different cost ratios in building the PCA-7 and PCA-18 models. By defining a cost ratio, we are trying to give greater importance to fault-prone modules. Recall that Type II misclassifications are more costly to the software development team than Type I. As the Tables 15 and 16 indicate, the difference in models' accuracies for different values of C_I/C_{II} is very significant. We chose the cost ratio $C_I/C_{II} = 0.54$ for model PCA-7 and $C_I/C_{II} = 0.65$ for model PCA-18. These values yield the most balanced misclassification rates for Release 1.

We are actively pursuing further experiments with the cost-ratio approach, in par-

Table 15: Impact of cost ratio on model PCA-7

	Misclassification rates							
	Release 1		Release 2		Release 3		Release 4	
C_I/C_{II}	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
0.10	94.269%	0.437%	94.989%	0.000%	96.422%	0.000%	99.331%	0.000%
0.20	80.760%	0.873%	82.964%	1.058%	87.121%	0.000%	93.670%	0.000%
0.30	62.807%	4.367%	66.772%	3.704%	70.635%	6.383%	78.487%	1.087%
0.40	45.292%	10.917%	47.363%	9.524%	49.828%	6.383%	59.701%	8.696%
0.50	30.702%	20.087%	30.432%	22.222%	33.200%	14.894%	37.133%	23.913%
0.51	29.152%	21.834%	28.718%	23.810%	31.654%	17.021%	35.461%	25.000%
0.52	27.895%	22.271%	27.453%	24.339%	30.080%	19.149%	33.505%	25.000%
0.53	26.404%	23.581%	26.398%	26.455%	28.792%	19.149%	31.652%	25.000%
0.54	**25.000%**	**24.891%**	**24.815%**	**29.101%**	**27.418%**	**19.149%**	**29.671%**	**28.261%**
0.55	24.006%	27.074%	23.708%	31.217%	26.188%	21.277%	27.972%	29.348%
0.56	22.865%	29.258%	22.442%	33.862%	25.272%	21.277%	26.402%	30.435%
0.57	21.696%	30.131%	21.071%	34.921%	23.984%	23.404%	25.270%	33.696%
0.58	20.409%	31.004%	19.910%	36.508%	22.696%	27.660%	23.675%	33.696%
0.59	19.357%	33.624%	18.565%	40.212%	21.523%	27.660%	22.440%	34.783%
0.60	18.538%	36.245%	17.352%	41.270%	20.235%	27.660%	20.844%	35.870%
0.65	14.064%	48.908%	12.131%	50.794%	14.911%	44.681%	14.308%	42.391%
0.70	9.181%	59.389%	8.386%	60.317%	10.675%	57.447%	9.367%	53.261%
0.80	3.889%	76.419%	3.296%	74.074%	4.207%	72.340%	3.320%	69.565%
0.90	1.170%	86.026%	0.949%	91.005%	1.345%	93.617%	1.261%	85.870%
1.00	0.205%	94.760%	0.475%	96.296%	0.401%	97.872%	0.386%	97.826%
2.00	0.000%	100.000%	0.000%	100.000%	0.000%	100.000%	0.000%	100.000%

Table 16: Impact of cost ratio on model PCA-18

Misclassification rates

	Release 1		Release 2		Release 3		Release 4	
C_I/C_{II}	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
0.10	96.608%	0.000%	98.312%	0.000%	99.056%	0.000%	99.974%	0.000%
0.20	88.567%	0.437%	90.770%	0.000%	94.419%	0.000%	98.456%	0.000%
0.30	77.982%	0.873%	82.068%	0.529%	87.121%	2.128%	95.008%	0.000%
0.40	65.526%	3.930%	70.121%	2.116%	75.930%	4.255%	86.284%	1.087%
0.50	51.637%	10.917%	53.745%	9.524%	61.649%	6.383%	69.557%	8.696%
0.55	43.801%	15.284%	45.464%	12.698%	52.948%	12.766%	60.602%	14.130%
0.60	36.053%	22.271%	37.131%	20.106%	45.621%	17.021%	51.004%	18.478%
0.61	34.357%	23.144%	35.575%	22.751%	44.076%	19.149%	49.279%	18.478%
0.62	33.012%	24.017%	33.993%	23.810%	42.788%	19.149%	47.427%	19.565%
0.63	31.608%	25.328%	32.437%	26.455%	41.299%	21.277%	45.625%	19.565%
0.64	30.526%	26.201%	30.881%	26.455%	39.611%	21.277%	43.773%	19.565%
0.65	**28.947%**	**27.511%**	**29.088%**	**26.984%**	**37.951%**	**21.277%**	**41.765%**	**20.652%**
0.66	27.485%	29.258%	27.927%	30.159%	35.861%	25.532%	39.732%	23.913%
0.67	26.082%	33.624%	26.187%	31.746%	33.572%	25.532%	37.082%	25.000%
0.68	24.561%	34.934%	24.947%	32.804%	31.626%	25.532%	35.461%	26.087%
0.69	22.865%	38.428%	23.391%	36.450%	29.737%	25.532%	33.479%	27.174%
0.70	21.637%	40.611%	22.073%	37.566%	27.962%	25.532%	31.446%	29.348%
0.80	8.538%	64.192%	8.412%	62.434%	10.933%	57.447%	11.554%	58.696%
0.90	1.813%	85.153%	1.714%	86.243%	2.290%	76.596%	2.290%	86.957%
1.00	0.322%	96.507%	0.158%	95.238%	0.372%	97.872%	0.309%	97.826%
2.00	0.000%	100.000%	0.000%	100.000%	0.000%	100.000%	0.000%	100.000%

Table 17: Comparison of cost ratio *vs.* scale factor

	Release 1		Release 2		Release 3		Release 4	
Model	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
Scale Factor								
PCA-5	31.1%	35.8%	31.8%	28.6%	33.8%	29.8%	41.5%	28.3%
PCA-7	34.3%	32.6%	28.5%	38.1%	29.7%	21.3%	34.3%	32.6%
PCA-10	32.1%	28.6%	32.1%	28.8%	33.9%	27.7%	38.1%	27.2%
PCA-18	25.6%	35.4%	26.7%	35.0%	27.4%	27.7%	36.1%	29.3%
Cost Ratio								
PCA-5	26.3%	27.1%	22.3%	32.8%	25.6%	29.8%	38.0%	28.3%
PCA-7	25.0%	24.9%	24.8%	29.1%	27.4%	19.1%	29.7%	28.3%
PCA-10	30.5%	31.4%	23.1%	41.3%	28.7%	36.2%	39.0%	26.1%
PCA-18	28.9%	27.5%	29.1%	27.0%	38.0%	21.3%	41.8%	19.1%

ticular, we are applying this approach to the standardized raw metrics models to complement the PCA-based results above. The further and more complete experiments with C_I/C_{II} are to be carried out in future research, where we will compare this approach to other methodologies.

Comparison with tree-based models. We compared PCA-based CBR models using the scale-factor approach to tree-based models. The CBR models used the same metrics that were selected as significant by the tree-based models in previous work [46]. Our hypothesis was that CBR using the same domain metrics as models from the previous research can obtain results that are not significantly worse than those obtained in the previous work [46]. If so, that would support the hypothesis that CBR is a robust tool that is competitive with other techniques..

Comparisons between PCA-5 and PCA-7 CBR-based models (scale-factor approach) and tree-based models, CART-10 and CART-18, built in [46] had showed that the tree-based models with PCA domain metrics performed significantly better than our PCA-5 and

Table 18: Comparison of CBR and tree-based models

	Misclassification rates							
	Release 1		Release 2		Release 3		Release 4	
Model	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
CART-10 [46]	30.4%	30.6%	26.6%	24.9%	28.8%	21.3%	32.7%	27.2%
CART-18 [46]	38.7%	22.3%	29.3%	21.2%	29.9%	19.1%	32.7%	19.6%
Scale Factor Approach								
PCA-5	31.1%	35.8%	31.8%	28.6%	33.8%	29.8%	41.5%	28.3%
PCA-7	34.3%	32.6%	28.5%	38.1%	29.7%	21.3%	34.3%	32.6%
Cost Ratio Approach								
PCA-5	26.3%	27.1%	22.3%	32.8%	25.6%	29.8%	38.0%	28.3%
PCA-7	25.0%	24.9%	24.8%	29.1%	27.4%	19.1%	29.7%	28.3%

This is a subset of Table 9 and Table 14.

PCA-7 models. We can see the evidence of this in Tables 25 through 30 in Appendix C. The tree-based models' accuracy is shown in Table 18. A possible reason for this may be the fact that we used equal weights for all the variables, except for *USAGE* in the CBR models. The results may be much more favorable for CBR models if we included appropriate weights with the corresponding metrics. Overall, PCA-5 and PCA-7 CBR-based models' accuracy remained acceptable given that uniform weights were used to determine the accuracy of these models. This may be a subject of further research.

4 Conclusions

Overall, after carrying out our experiments, we observed that, generally, CBR models using scale factors did not gain in performance from any of the following:

- Replacing the raw metrics with the most significant raw metrics, selected by tree-based models.

- Replacing the raw metrics with the PCA-generated metrics.
- Replacing the PCA-generated metrics with most significant PCA-generated metrics, selected by tree-based models.
- Adding the process metrics to the models that are using only product and execution metrics.

The above conclusions indicate that CBR methodology based on scale factors may not benefit from prior selection of the most significant metrics and is capable of producing acceptable results with raw metrics without having to convert them into PCA domains. However, many analysts would prefer the more parsimonious models (fewer independent variables) on general principles. Moreover, when we have a large number of independent variables, the computation of misclassification rates takes more time and computing resources. In that case, it is useful to carry out some kind of selection of the most significant metrics in order to allow for faster processing and and to save computing resources.

However, the CBR methodology based on cost ratios generally produced models that were more accurate than corresponding scale factor models. The disadvantages listed above for the scale-factor approach may not apply to the cost-ratio approach. For example, adding process metrics did improve cost-ratio models. The cost-ratio methodology deserves further investigation.

Although our comparisons of CBR models versus the tree-based models built in [46] and [40] were not always advantageous, we were able, for some cases achieve similar results.

Our use of uniform weights for all metrics, except for *USAGE*, suggests that we have a lot of areas for future research and improvement.

Acknowledgments

We thank Ken McGill for his encouragement and support. We thank Bojan Cukic for helpful discussions. This work was supported in part by Cooperative Agreement NCC 2-1141 from NASA Ames Research Center, Software Technology Division (Independent Verification and Validation Facility). The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor.

References

[1] Y. Berkovich. Software quality prediction using case-based reasoning. Master's thesis, Florida Atlantic University, Boca Raton, Florida USA, Aug. 2000. Advised by Taghi M. Khoshgoftaar.

[2] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044, Nov. 1993.

[3] J. Deng. Classification of software quality using tree modeling with the S-Plus algorithm. Master's thesis, Florida Atlantic University, Boca Raton, Florida, Dec. 1999. Advised by Taghi M. Khoshgoftaar.

[4] W. R. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications.* John Wiley & Sons, New York, 1984.

[5] C. Ebert. Classification techniques for metric-based software development. *Software Quality Journal*, 5(4):255–272, Dec. 1996.

[6] U. M. Fayyad. Data mining and knowledge discovery: Making sense out of data. *IEEE Expert*, 11(4):20–25, Oct. 1996.

[7] K. Ganesan, T. M. Khoshgoftaar, and E. B. Allen. Case-based software quality prediction. *International Journal of Software Engineering and Knowledge Engineering*, 9(6), 1999. In press.

[8] M. H. Halstead. *Elements of Software Science.* Elsevier, New York, 1977.

[9] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. EMERALD: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, Sept. 1996.

[10] W. D. Jones, J. P. Hudepohl, T. M. Khoshgoftaar, and E. B. Allen. Application of a usage profile in software quality models. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 148–157, Amsterdam, Netherlands, Mar. 1999. IEEE Computer Society.

[11] B. P. Kettler, J. A. Hendler, and M. P. Evett. Massively parallel support for a case-based planning system. *IEEE Expert*, 9(1):8–14, Feb. 1994.

[12] T. M. Khoshgoftaar and E. B. Allen. Classification techniques for predicting software quality: Lessons learned. In *Proceedings of the Annual Oregon Workshop on Software Metrics*, Coeur d'Alene, Idaho USA, May 1997. University of Idaho.

[13] T. M. Khoshgoftaar and E. B. Allen. Predicting the order of fault-prone modules in legacy software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 344–353, Paderborn, Germany, Nov. 1998. IEEE Computer Society.

[14] T. M. Khoshgoftaar and E. B. Allen. The stability of software quality models over multiple releases. Technical Report TR-CSE-98-25, Florida Atlantic University, Boca Raton, Florida USA, Nov. 1998.

[15] T. M. Khoshgoftaar and E. B. Allen. A comparative study of ordering and classification of fault-prone software modules. *Empirical Software Engineering: An International Journal*, 4:159–186, 1999.

[16] T. M. Khoshgoftaar and E. B. Allen. Modeling the risk of software faults. Technical Report TR-CSE-00-06, Florida Atlantic University, Boca Raton, Florida USA, Feb. 2000.

[17] T. M. Khoshgoftaar and E. B. Allen. A practical classification rule for software quality models. *IEEE Transactions on Reliability*, 49(2), June 2000. In press.

[18] T. M. Khoshgoftaar, E. B. Allen, L. A. Bullard, R. Halstead, and G. P. Trio. A tree-based classification model for analysis of a military software system. In *Proceedings of the IEEE High-Assurance Systems Engineering Workshop*, pages 244–251, Niagara on the Lake, Ontario, Canada, Oct. 1996. IEEE Computer Society.

[19] T. M. Khoshgoftaar, E. B. Allen, and J. C. Busboom. SMART: Software measurement analysis and reliability toolkit. Technical Report TR-CSE-98-21, Florida Atlantic University, Boca Raton, FL USA, July 1998.

[20] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, and G. P. Trio. Detection of fault-prone software modules during a spiral life cycle. In *Proceedings of the International Conference on Software Maintenance*, pages 69–76, Monterey, CA, Nov. 1996. IEEE Computer Society.

[21] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. Flass. Process measures for predicting software quality. In *Proceedings of the IEEE High-Assurance Systems Engineering Workshop*, Washington, DC, Aug. 1997. IEEE Computer Society.

[22] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Which software modules have faults that will be discovered by customers? Technical Report TR-CSE-97-55, Florida Atlantic University, Boca Raton, FL, Aug. 1997.

[23] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Which software modules have faults that will be discovered by customers? *Journal of Software Maintenance: Research and Practice*, 11(1):1–18, Jan. 1999.

[24] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 6, 2000. In press.

[25] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, Jan. 1996.

[26] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. The impact of software evolution and reuse on software quality. *Empirical Software Engineering: An International Journal*, 1(1):31–44, 1996.

[27] T. M. Khoshgoftaar, E. B. Allen, A. Naik, W. D. Jones, and J. P. Hudepohl. Using classification trees for software quality models: Lessons learned. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 82–89, Bethesda, Maryland USA, Nov. 1998. IEEE Computer Society.

[28] T. M. Khoshgoftaar, E. B. Allen, and R. Shan. Benefits of principal components analysis with classification trees of fault-prone software modules. In H. Pham and M.-W. Lu, editors, *Proceedings: Sixth ISSAT International Conference on Reliability and Quality in Design*, Orlando, Florida USA, Aug. 2000. International Society of Science and Applied Technologies. Invited paper. In press.

[29] T. M. Khoshgoftaar, E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl. Assessing uncertain predictions of software quality. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 159–168, Boca Raton, Florida USA, Nov. 1999. IEEE Computer Society.

[30] T. M. Khoshgoftaar, K. Ganesan, E. B. Allen, F. D. Ross, R. Munikoti, N. Goel, and A. Nandi. Predicting fault-prone modules with case-based reasoning. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 27–35, Albuquerque, New Mexico USA, Nov. 1997. IEEE Computer Society.

[31] T. M. Khoshgoftaar and D. L. Lanning. An alternative modeling approach for predicting program changes. *Computer Science and Informatics: CSI Journal*, 25(3):25–38, Sept. 1995.

[32] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya. A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. *IEEE Journal on Selected Areas in Communications*, 12(2):279–291, Feb. 1994.

[33] T. M. Khoshgoftaar and R. M. Szabo. Improving code churn predictions during the system test and maintenance phases. In *Proceedings of the International Conference on Software Maintenance*, pages 58–67, Victoria, BC Canada, Sept. 1994. IEEE Computer Society.

[34] J. L. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[35] R. Kowalski. AI and software engineering. In *Artificial Intelligence and Software Engineering*, pages 339–352. Ablex Publishing, Norwood, NJ USA, 1991.

[36] F. Lanubile. Why software reliability predictions fail. *IEEE Software*, 13(4):131–132,137, July 1996.

[37] D. B. Leake, editor. *Case-Based Reasoning: Experiences, Lessons, and Future Directions.* MIT Press, Cambridge, MA USA, 1996.

[38] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE–2(4):308–320, Dec. 1976.

[39] G. J. Myers. *Composite/Structured Design.* Van Nostrand Reinhold, New York, 1978.

[40] A. Naik. Prediction of software quality using classification tree modeling. Master's thesis, Florida Atlantic University, Boca Raton, FL USA, Dec. 1998. Advised by Taghi M. Khoshgoftaar.

[41] R. J. Offen and R. Jeffery. Establishing software measurement programs. *IEEE Software*, 14(2):45–53, Mar. 1997.

[42] S. L. Pfleeger. Assessing measurement. *IEEE Software*, 14(2):25–26, Mar. 1997. Editor's introduction to special issue.

[43] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, Mar. 1990.

[44] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, May 1992.

[45] G. A. F. Seber. *Multivariate Observations*. John Wiley and Sons, New York, 1984.

[46] R. Shan. Modeling software quality with classification trees using principal components analysis. Master's thesis, Florida Atlantic University, Boca Raton, Florida, Dec. 1999. Advised by Taghi M. Khoshgoftaar.

[47] N. T. Smith and K. Ganesan. Software design using case-based reasoning. In *Proceedings of the Fourth Software Engineering Research Forum*, pages 193–200, Boca Raton, FL, Nov. 1995.

[48] C. Vasudevan and K. Ganesan. Case-based path planning for autonomous underwater vehicles. In *Proceedings of the Ninth International Symposium on Intelligent Control*, pages 160–165. IEEE, Aug. 1994.

Appendices

A Software Measurement Analysis and Reliability Toolkit

The Software Measurement Analysis and Reliability Toolkit (SMART) is a domain-independent research tool for case-based reasoning and other modeling techniques, whose user interface is tailored for software quality modeling. An "analysis project" is our term for the analyst's task of building and using a software quality model for the benefit of a software development project.

SMART was developed in 1998 with support from the Empirical Software Engineering Laboratory at Florida Atlantic University, and industrial partners. Details are

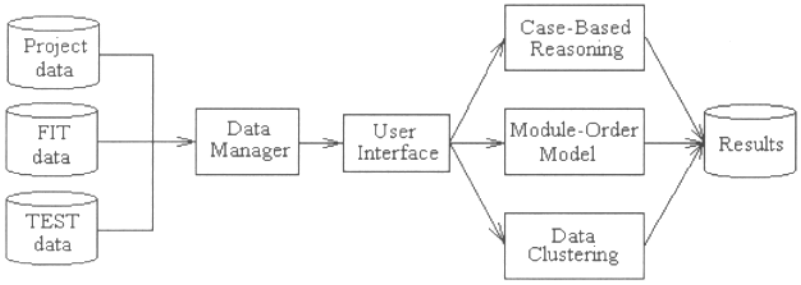

Figure 1: Tool architecture

documented in [19].

A.1 Toolkit Architecture

The software architecture of SMART is shown in Figure 1. The tool is implemented using Microsoft Visual C++® and runs in a Windows 95® or Windows/NT® environment. There is a central data manager that handles the *fit* data, which is used to build the model, the *test* data, which is used to validate the model, and analysis-project data. The models are controlled via the graphical user interface and results may be in the form of a display, printed report, or data file.

The user interface is based on a "dialog-based property sheet". Figure 2 shows the General page. Tabs for the various types of models are along the top portion of the property sheet, while the current page is displayed below the tabs. Navigation through the menus and selection of the various available options is done with a mouse. Once satisfied with the options selected, the analyst may save the analysis project's data, run an experiment, or print out statistics about the current analysis project. There is also an online help document that explains how to use the tool.

There are currently three types of models supported by SMART. The first type

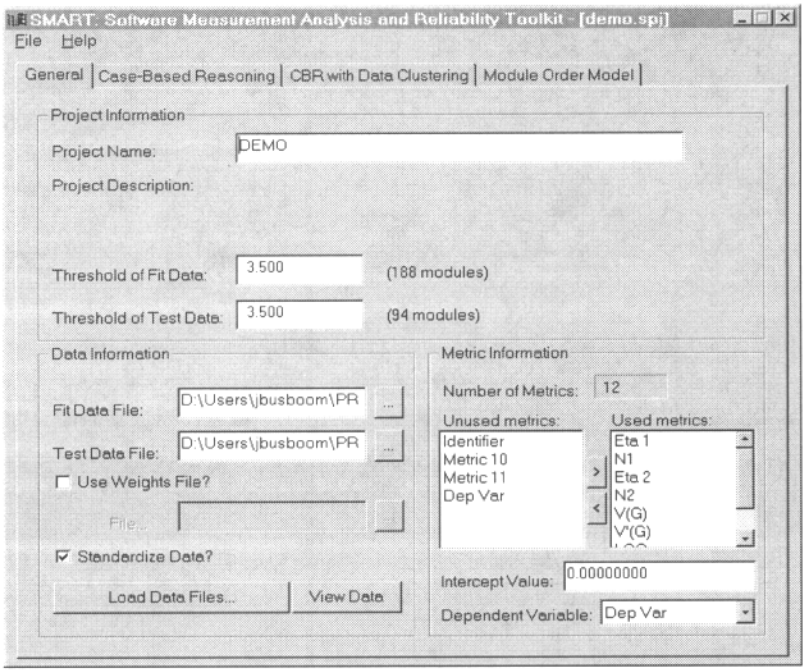

Figure 2: Project data

of model is for classification based on case-based reasoning (CBR). This kind of model classifies a software module as *fault-prone* or not by comparing its attributes to a library of similar past modules. The second model extends the CBR classification model with cluster analysis [31]. This technique partitions the case library into clusters. An unclassified module is classified according to the closest cluster. The third type of model is the module-order model [13, 15], which predicts the rank-order of modules according to a quantitative dependent variable. It can also be used for classification. The module-order model is not discussed here.

On the General page, the analyst specifies the Project Name and Project Description of the analysis project. Modeling parameters and results can be saved under this name. The analyst specifies the names of space-delimited ASCII files holding the data (Fit Data

File and Test Data File). The names of fields are required as the first line of a data file, and each line thereafter is data for one software module. The *fit* data set consists of data from a past software development project including values of both dependent and independent variables. It is used to build a model and estimate model parameters, if any.

The *test* data set is used to simulate use of the model. It also consists of data from a past software development project. The independent variable values are used to make predictions as if the data were from a current software development project, and the actual dependent variable values are used to evaluate the accuracy of the predictions.

After a model is built and evaluated it can be used to predict the classes of a *current* data set. The Test Data File can be data from a current software development project whose independent variable values are known, but whose dependent variable values are not. Let i and j be indices for modules in data sets.

One of the data fields should be a software quality factor that the analyst identifies as the Dependent Variable, such as the number of faults. Let y_i be the actual value of the dependent variable for module i, and let $\hat{y}_i$ be its predicted value.

In our application, software metrics are the independent variables. The analyst selects the independent variables, indexed by $k = 1, \ldots, m$. Variables included in the model are listed as Used metrics, and those in the data file but not in the model are listed as Unused metrics.

The analyst can choose to standardize the independent variables so that all have the same unit of measure. Given a raw measurement, x_{ik}, its estimated mean, $\bar{x}_k$, and its estimated standard deviation, s_k, the standardized measurement is $z_{ik} = (x_{ik} - \bar{x}_k)/s_k$. The unit of measure is a standard deviation. In the discussion below, we use x_{ik} to mean either a raw or a standardized measurement, as selected by the analyst.

The actual class of a module is defined by whether or not the dependent variable is greater than a Threshold, θ, or not. The analyst specifies thresholds for the fit and test data sets. In many of our case studies, we use the same threshold for both. However, the tool provides additional flexibility. The actual class of a module is given by

$$Class_i = \begin{cases} not\ fault\text{-}prone & \text{if } y_i < \theta \\ fault\text{-}prone & \text{Otherwise} \end{cases} \tag{6}$$

Not all software metrics are equally related to software faults. It is possible that small changes in one attribute may strongly relate to the number of faults. The analyst can specify a Weights File containing weights, w_k, for each independent variable in the model to account for various levels of importance. An Intercept Value, w_0, can also be specified on this page.

A.2 Case-Based Reasoning Features

The working hypothesis for a software quality model is this: a module currently under development is probably *fault-prone* if a module with similar attributes in an earlier version or similar project was considered to be *fault-prone*. Figure 3 depicts the SMART's Case-Based Reasoning page.

The analyst specifies the data set to be used as the Case Library, usually the fit data set. A module in the library is a *case*. Let c_{jk} be the value of the k^{th} independent variable for case j, and let $\mathbf{c}_j$ be the vector of independent variable values for case j.

The analyst also specifies the Target Data Set whose dependent variable values are to be predicted. Let x_{ik} be the value of the k^{th} independent variable for target module i, and let $\mathbf{x}_i$ be the vector of independent variable values for module i. When using the

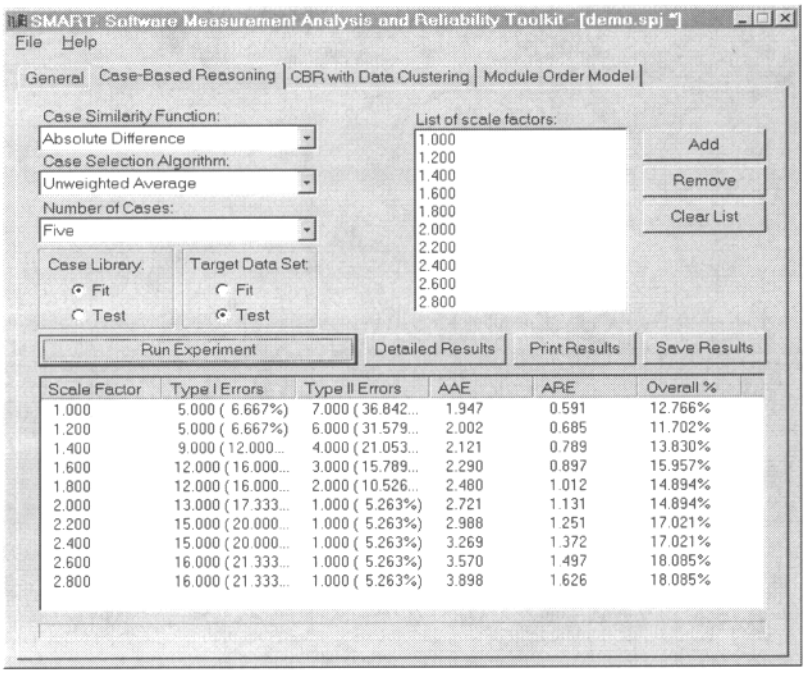

Figure 3: Case-based reasoning model

model, the current set of modules is the target. When evaluating the model, the test data set is the target. When experimenting with parameter values during model development, the fit data set can be designated as the target.

The analyst selects a Case Similarity Function which calculates a distance, d_{ij}, between the current target module $\mathbf{x}_i$ and every case $\mathbf{c}_j$, where a small distance implies the modules are very similar. Based on the distances, SMART identifies a set of cases who are "nearest neighbors" to the current module, $\mathbf{x}_i$. The analyst selects the Number of Cases in this set. The analyst also selects a *solution algorithm* (Case Selection Algorithm) which predicts the dependent variable, $\hat{y}_i$ of the current module, $\mathbf{x}_i$, and predicts its class.

The analyst can provide a List of scale factors, α, for empirical investigation. Each value of α represents a distinct experiment over all target modules. Summary statistics

for the experiments are listed, as shown in Figure 3.

A Type I misclassification is when a model classifies a module as *fault-prone* which is actually *not fault-prone*, and a Type II misclassification is when a model classifies a module as *not fault-prone* which is actually *fault-prone*. SMART provides counts and percentages. SMART summarizes the accuracy of the predicted dependent variable by average absolute error (AAE) and average relative error (ARE) which are given by

$$AAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \quad (7)$$

$$ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|\hat{y}_i - y_i|}{y_i + 1} \quad (8)$$

where n is the number of modules in the data set. The denominator of ARE has one added to avoid division by zero. The Overall misclassification rate is also provided.

Similarity Functions Several measures of similarity are presented in the literature according to the problem domain, the availability of attribute data, and whether data types are categorical, discrete, real, etc. [34]. For quantitative attributes, the city-block distance and Euclidean distance [30] are commonly used. The Mahalanobis distance [4] has the advantage of explicitly accounting for correlations among attributes. SMART supports the following similarity functions.

Absolute Difference distance is also known as "city-block" distance or "Manhattan" distance. The distance is the weighted sum of the absolute value of the difference in independent variables between a module and a case. The weights are those provided by the analyst. The absolute value is taken because distance is computed irrespective of direction.

$$d_{ij} = \sum_{k=1}^{m} w_k\,|c_{jk} - x_{ik}| \quad (9)$$

Euclidean Distance considers each independent variable as a dimension of an m-dimensional space. A module is represented by a point in this space. The Euclidean distance between a module and a case is their weighted distance in this space. The weights are those provided by the analyst.

$$d_{ij} = \left(\sum_{k=1}^{m} (w_k (c_{jk} - x_{ik}))^2 \right)^{1/2} \tag{10}$$

Linear Regression 1 is intended for when the weights, w_k, supplied by the analyst were estimated by multiple linear regression of the dependent variable as a function of case attributes, $\mathbf{c}_j$, using some other tool, such as a spreadsheet or a statistical modeling tool. The weights are used in a linear model.

$$\hat{f}(\mathbf{c}_j) = w_0 + w_1 c_{j1} + \ldots + w_m c_{jm} \tag{11}$$

The distance is the difference in the dependent variable values predicted by the linear model for the attributes of the module and the case.

$$d_{ij} = \hat{f}(\mathbf{c}_j) - \hat{f}(\mathbf{x}_i) = \sum_{k=1}^{m} w_k (c_{jk} - x_{ik}) \tag{12}$$

Linear Regression 2 is also intended for when the weights, w_k, supplied by the analyst were estimated by multiple linear regression of the dependent variable as a function of case attributes, $\mathbf{c}_j$. The distance is the difference between the actual y_j of the case and the predicted $\hat{y}_i = \hat{f}(\mathbf{x}_i)$ of the module.

$$d_{ij} = y_j - \hat{f}(\mathbf{x}_i) = y_j - \sum_{k=1}^{m} w_k x_{ik} - w_0 \tag{13}$$

Mahalanobis Distance [4] is an alternative to Euclidean distance in cases where metrics may be poorly scaled or highly correlated. The distance is given by

$$d_{ij} = (\mathbf{c}_j - \mathbf{x}_i)' \, \mathbf{S}^{-1} \, (\mathbf{c}_j - \mathbf{x}_i) \tag{14}$$

where prime ($'$) means transpose, and $\mathbf{S}$ is the covariance matrix of the independent variables over all of the case library, and $\mathbf{S}^{-1}$ is its inverse. In the special case where the independent variables are uncorrelated and the variances are all the same, $\mathbf{S}$ is the identity matrix and the Mahalanobis distance is the Euclidean distance squared.

Nearest Neighbors After the tool calculates distances between a module $\mathbf{x}_i$ and all cases using one of the similarity functions, the distances are sorted. The cases, $\mathbf{c}_j$, with the smallest distances, d_{ij}, are of primary interest. The set of nearest neighbors, $\mathcal{N}$, is an input to each solution algorithm below. The analyst selects the number of nearest neighbors, $n_{\mathcal{N}} \in \{1, 3, 5, 7, 9\}$. Based on a preliminary empirical investigation with software quality data, this range appears to be adequate.

Solution Algorithms Each solution algorithm assigns the unclassified module to a class. Predictions of the dependent variable made by the solution algorithm can be scaled in order to improve the accuracy of class predictions. Most of the solution algorithms below calculate $\hat{y}_i$ and then classify the module by

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{If } \alpha\hat{y}_i < \theta \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \tag{15}$$

To choose a preferred value of α, the analyst designates the fit data set as the case library and also as the target data set, and supplies a list of candidate scale factors, α. The analyst can then choose a preferred value of α based on experiment results. When the target is the test data set and when the target is a current data set, the analyst can then specify the preferred scale factor, α.

SMART supports the following solution algorithms.

Unweighted average. This solution algorithm averages the dependent variable, y_j, of the closest $n_{\mathcal{N}}$ modules from the case library to form a value of $\hat{y}_i$ for the target module.

$$\hat{y}_i = \frac{1}{n_{\mathcal{N}}} \sum_{j \in \mathcal{N}} y_j \tag{16}$$

We are primarily concerned with classification, so the value predicted is not as important as the predicted class, given by Equation (15).

Inverse-distance weighted average. This solution algorithm utilizes the distance measures for the $n_{\mathcal{N}}$ closest cases as weights in a weighted average. Because a smaller distance means a better match, SMART weights each case in the nearest-neighbor set by a normalized inverse distance, δ_{ij}.

$$\delta_{ij} = \frac{1/d_{ij}}{\sum_{j \in \mathcal{N}} 1/d_{ij}} \tag{17}$$

$$\hat{y}_i = \sum_{j \in \mathcal{N}} \delta_{ij}\, y_j \tag{18}$$

The case that is most similar to the target module will naturally have the largest weight, and therefore play a larger role in the classification of the module by Equation (15).

Rank-weighted average. In this solution algorithm, the nearest neighbors are ranked according to their distances from the module. Rank $R_j = 1$ represents the best match while $R_j = n_{\mathcal{N}}$ represents the worst match among the nearest neighbors. The rank-weight, ρ_{ij}, is given by

$$\rho_{ij} = \frac{n_{\mathcal{N}} - R_j + 1}{\sum_{j \in \mathcal{N}} R_j} \tag{19}$$

$$\hat{y}_i = \sum_{j \in \mathcal{N}} \rho_{ij}\, y_j \tag{20}$$

The module is classified by Equation (15).

Majority vote. This solution algorithm assigns the unclassified module to the class of the majority of the cases in the nearest-neighbor set.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if majority of } y_j < \theta \text{ for } j \in \mathcal{N} \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \tag{21}$$

A.3 Data Clustering

In addition to the basic CBR model, SMART provides an extension by cluster analysis to enhance classification. The case library is partitioned into clusters according to the actual class of each case. SMART compares a module to the *fault-prone* cluster and the *not fault-prone* cluster and determines the closest group. SMART supports the same similarity functions and solution algorithms here as in the CBR model.

The Data Clustering page (DC) in Figure 4 is similar to the CBR page. Instead of scale factors, this page provides for a List of cost ratios, C_I/C_{II}. The analyst can calculate a cost ratio as the ratio of the cost of a Type I misclassification, C_I, to the cost of a Type II misclassification, C_{II}. However, this is often difficult to estimate, and therefore, SMART provides for experiments with a list of values. In software quality modeling, a Type II misclassification can be much more serious than a Type I, because the cost of releasing fault-prone modules to analysts is usually much more expensive than wasting effort enhancing low-risk modules. The analyst can experiment to choose a preferred C_I/C_{II} value.

The analyst has the option of using a Pooled Covariance matrix, $\mathbf{S}$, for the Mahalanobis distance measure. "Pooled" means that all data points from both clusters are used in determining the $\mathbf{S}$ matrix, and "non-pooled" means that two individual $\mathbf{S}$ matrices are calculated for the *fault-prone* cluster and the *not fault-prone* cluster.

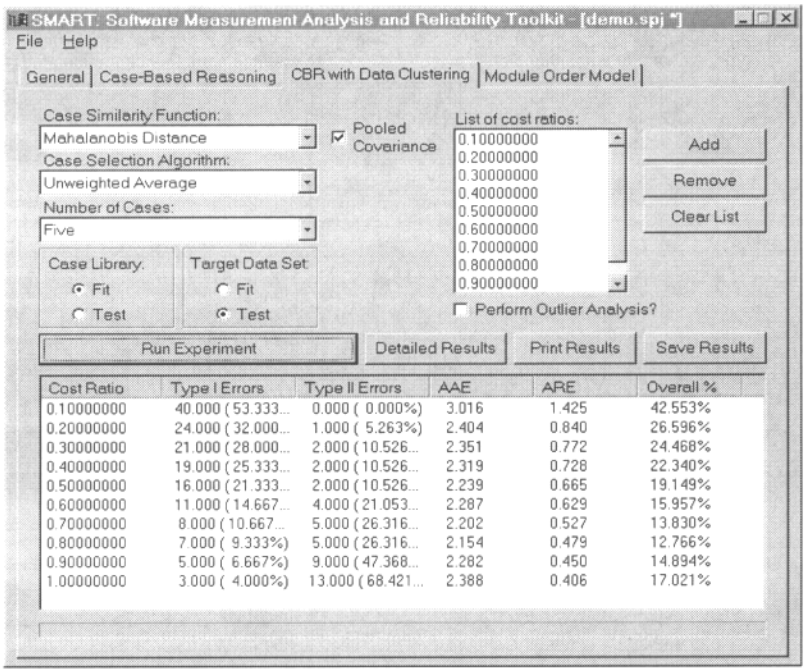

Figure 4: Data clustering model

The classification rule used here is based on our recent work with statistical classification techniques [23]. For an unclassified module, $\mathbf{x}_i$, let $d_{nfp}(\mathbf{x}_i)$ be the average distance to *not fault-prone* nearest-neighbor cases, and let $d_{fp}(\mathbf{x}_i)$ be the average distance to *fault-prone* nearest-neighbor cases. The module is classified by

$$Class(\mathbf{x}_i) = \begin{cases} not\ fault\text{-}prone & \text{If } \frac{d_{fp}(\mathbf{x}_i)}{d_{nfp}(\mathbf{x}_i)} > \frac{C_I}{C_{II}} \\ fault\text{-}prone & \text{Otherwise} \end{cases} \tag{22}$$

where C_I/C_{II} is experimentally chosen. The analyst can choose C_I/C_{II} by a similar method as α, described above.

The Data Clustering page also offers Outlier Analysis. An outlier is a case that has abnormal attributes. One module at a time is removed from the case library and its class is predicted. If both the actual and predicted classes are the same, then the module is

not an outlier, otherwise, it is marked as an outlier and is not used as part of the case library. This technique is repeatedly applied to all modules from the case library until all outliers are removed.

B Principal Components Analysis

The following description of principal components analysis (PCA) is taken from [20]. Software metrics have a variety of units of measure, which are not readily combined in a multivariate model. We transform all metric variables, so that each standardized variable has a mean of zero and a variance of one. Thus, the common unit of measure becomes one standard deviation.

Principal components analysis is a statistical technique for transforming multivariate data into orthogonal variables, and for reducing the number of variables without losing significant variation. Suppose we have m measurements on n modules. Let $\mathbf{Z}$ be the $n \times m$ matrix of standardized measurements where each row corresponds to a module and each column is a standardized variable. Our principal components are linear combinations of the m standardized random variables, $Z_1, \ldots, Z_m$. The principal components represent the same data in a new coordinate system, where the variability is maximized in each direction and the principal components are uncorrelated [45]. If the covariance matrix of $\mathbf{Z}$ is a real symmetric matrix with distinct roots, then one can calculate its eigenvalues, λ_k, and its eigenvectors, $\mathbf{e}_k, k = 1, \ldots, m$. Since the eigenvalues form a nonincreasing series, $\lambda_1 \geq \ldots \geq \lambda_m$, one can reduce the dimensionality of the data without significant loss of explained variance by considering only the first p components, $p \ll m$, according to some stopping rule, such as achieving a threshold of explained variance. For example,

choose the minimum p such that $\sum_{k=1}^{p} \lambda_k / m \geq 0.90$ to achieve at least 90% of explained variance.

Let $\mathbf{T}$ be the $m \times p$ standardized transformation matrix whose columns, $\mathbf{t}_j$, are defined as

$$\mathbf{t}_k = \frac{\mathbf{e}_k}{\sqrt{\lambda_k}} \text{ for } k = 1, \ldots, p \tag{23}$$

Let D_k be a principal component random variable, and let $\mathbf{D}$ be an $n \times p$ matrix with D_k values for each column, $k = 1, \ldots, p$.

$$D_k = \mathbf{Z}\mathbf{t}_k \tag{24}$$

$$\mathbf{D} = \mathbf{Z}\mathbf{T} \tag{25}$$

When the underlying data is software metric data, we call each D_k a *domain metric.*

C Detailed Empirical Results[1]

C.1 Experiments with Selected Raw Metrics

We experimented to see whether we can achieve good results by performing CBR on the data sets using only the variables that were determined to be significant in tree-based models built by CART [40]. We had performed the experiments for the four product metrics and for 11 product and process metrics. During these experiments, we varied the weights for *USAGE*. We wanted to see how the model built by SMART will compare against the CART model using the same variables.

Our hypothesis was that CBR models built using SMART will perform better or similar to the performance of the models built using tree methodology in previous research.

Table 19: Raw-4 Model Release 2

weight of *USAGE*	7 cases		9 cases	
	Type I	Type II	Type I	Type II
1	27.716%	35.979%	32.199%	31.746%
2	27.347%	32.275%	32.173%	27.513%
3	26.477%	31.746%	31.355%	26.455%
4	25.976%	32.804%	31.382%	25.926%
5	25.923%	31.217%	30.459%	26.459%
6	25.923%	33.333%	30.617%	26.455%
7	26.055%	33.333%	30.643%	26.455%
8	26.002%	31.217%	**30.643%**	**24.868%**
9	26.160%	32.275%	30.152%	26.455%
10	26.055%	32.275%	30.432%	26.984%

Table 19 displays the results of our experiments with the four significant product metrics chosen in [40] for Release 2. Based on the results obtained from the experiments with RAW-4 model for Release 2, we determined that the best model was obtained with 9 similar cases and with the weight of *USAGE* being 8. The scale factor for this model was 9.00. This model seems to have the most balanced misclassification rates and one of the lowest Type II misclassification rates. Therefore, we will consider the performance of this model in Releases 3, and 4 as the best model for our system. Table 20 displays the results of our experiments with the 4 significant product metrics chosen in [40] for Release 3. Next, Table 21 displays the results of our experiments with the 4 significant product metrics chosen in [40] for Release 4.

The three tables, Table 22, Table 23, and Table 24 display the results of our experiments with the 11 significant product and process metrics chosen in [40] for Releases 2, 3, and 4. Likewise, as can be seen from the Table 22, the best model was found to be the one with 9 similar cases and the weight of *USAGE* equal 6 (with scale factor of 9.00).

Table 20: Raw-4 Model Release 3

weight	7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II
1	28.048%	31.915%	32.284%	23.404%
2	27.418%	34.043%	32.313%	25.532%
3	27.333%	34.043%	31.568%	21.277%
4	27.075%	31.915%	31.597%	21.277%
5	27.275%	31.915%	31.597%	19.149%
6	27.132%	34.043%	31.797%	21.277%
7	27.390%	34.043%	31.712%	21.277%
8	27.562%	31.915%	**32.225%**	**21.277%**
9	27.848%	27.660%	32.284%	23.404%
10	27.790%	27.660%	32.055%	23.404%

Table 21: Raw-4 Model Release 4

weight	7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II
1	33.453%	34.783%	38.806%	25.000%
2	33.402%	34.783%	38.883%	30.435%
3	33.685%	33.696%	39.115%	30.435%
4	33.942%	32.609%	39.063%	27.174%
5	34.071%	34.783%	39.629%	27.174%
6	34.328%	33.696%	39.707%	28.261%
7	34.568%	32.609%	39.707%	28.261%
8	34.817%	31.522%	**39.604%**	**29.348%**
9	34.766%	31.522%	39.398%	29.348%
10	34.843%	31.522%	39.372%	29.348%

The results of the prior research using a tree-based model [40] can be found in rows 1 and 2 of Table 9 in the body of this paper. The CBR models and the tree-based models performed very similarly for 11 product and process variables. The same is true for CBR RAW-4 model's comparison with the tree-based model for 3 product metrics and *USAGE*. The Tree-based models did perform better for the Release 4 but not for Releases 2 and 3. All four models in this experiment performed acceptably.

Table 22: Raw-11 Model Release 2

weight	5 cases		7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II	Type I	Type II
1	21.941%	39.683%	26.793%	31.217%	31.487%	25.926%
2	22.099%	42.857%	27.954%	31.746%	32.410%	26.455%
3	22.126%	43.386%	27.136%	31.746%	32.226%	25.926%
4	21.862%	42.328%	27.558%	33.862%	32.015%	28.042%
5	21.730%	42.328%	27.136%	35.450%	31.514%	25.397%
6	21.414%	41.799%	26.820%	33.862%	**31.461%**	**24.868%**
7	21.203%	40.212%	26.477%	33.333%	31.250%	26.984%
8	21.123%	39.683%	26.530%	32.804%	31.039%	25.926%
9	21.255%	40.741%	26.661%	33.333%	30.723%	26.455%
10	21.123%	40.212%	26.582%	32.275%	30.934%	27.513%

Table 23: Raw-11 Model Release 3

weight	5 cases		7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II	Type I	Type II
1	23.125%	36.170%	29.078%	27.660%	33.772%	21.277%
2	23.469%	31.915%	29.164%	25.532%	33.515%	25.532%
3	23.011%	38.298%	28.678%	23.404%	32.799%	23.404%
4	23.039%	31.915%	28.163%	23.404%	32.627%	19.149%
5	23.211%	27.660%	27.905%	21.277%	32.713%	19.149%
6	23.383%	25.532%	28.248%	23.404%	**32.713%**	**21.277%**
7	23.240%	27.660%	28.392%	21.277%	32.856%	19.149%
8	23.154%	27.660%	28.363%	19.149%	32.999%	14.894%
9	23.383%	25.532%	28.449%	14.894%	33.314%	14.894%
10	23.440%	25.532%	28.563%	17.021%	33.228%	17.021%

C.2 Comparisons of Models With Selected PCA Metrics and Models Using All PCA Metrics

The purpose was to determine how CBR models for these variables compare to the CBR models using all PCA domain variables.

Our hypothesis was that model PCA-10 would perform similarly or better than the

Table 24: Raw-11 Model Release 4

weight	5 cases		7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II	Type I	Type II
1	27.406%	47.826%	34.303%	34.783%	14.025%	47.826%
2	27.226%	45.652%	34.097%	35.870%	39.629%	30.435%
3	28.178%	45.652%	34.766%	34.783%	40.144%	30.435%
4	28.718%	43.478%	35.100%	32.609%	40.556%	32.609%
5	28.899%	43.478%	35.409%	33.696%	41.148%	31.522%
6	29.233%	42.391%	35.666%	34.783%	**40.968%**	**30.435%**
7	29.027%	41.304%	35.589%	33.696%	41.276%	29.348%
8	29.362%	39.130%	36.258%	32.609%	41.148%	26.087%
9	29.516%	38.043%	36.155%	32.609%	41.405%	27.174%
10	29.465%	39.130%	36.130%	32.609%	41.431%	23.913%

model PCA-5 whose independent variables were selected in the previous work [46]. Likewise, we expected that model PCA-18 would perform similarly or better than the model PCA-7 whose independent variables were selected in the previous work. If so, this would support the hypothesis that CBR does not need any pre-selection of metrics to create an effective quality prediction model, which again would indicate the usefulness of CBR as a quality prediction methodology.

Table 25, Table 26, and Table 27 display the results of our experiments with six significant product and process PCA domain metrics as well as *USAGE* that were selected in [46] for Releases 2, 3, and 4. Based on the results of the Release 2, we were able to choose the best model, which was the one using 9 similar cases with the weight of 10 for *USAGE* (with a scale factor α of 9.00). The model we chose also had the lowest cross-validation misclassification rate (the tables for Release 1 models were not included here). After the measurements of the model performance, we discovered that we, indeed made the right choice, since the CBR model did perform better for releases 3 and 4.

Table 25: PCA-7 Model Release 2

weight of *USAGE*	7 cases		9 cases	
	Type I	Type II	Type I	Type II
1	24.789%	43.386%	28.270%	39.153%
2	24.535%	43.915%	29.246%	35.450%
3	24.288%	44.974%	28.534%	38.624%
4	24.499%	45.503%	28.797%	38.624%
5	24.552%	47.090%	28.771%	40.741%
6	24.420%	48.148%	28.850%	41.270%
7	24.209%	49.206%	28.745%	42.857%
8	23.998%	47.619%	28.613%	39.683%
9	24.077%	48.677%	28.613%	39.683%
10	24.051%	48.148%	**28.507%**	**38.095%**
11	23.734%	47.090%	28.428%	38.624%

Table 26: PCA-7 Model Release 3

weight of *USAGE*	7 cases		9 cases	
	Type I	Type II	Type I	Type II
1	25.587%	36.170%	29.679%	31.915%
2	25.444%	34.043%	29.994%	27.660%
3	25.100%	34.043%	29.336%	23.404%
4	25.472%	31.915%	29.966%	25.532%
5	25.186%	29.787%	29.622%	21.277%
6	24.814%	29.787%	29.136%	21.277%
7	24.957%	34.043%	29.565%	23.404%
8	24.814%	34.043%	29.508%	25.532%
9	25.014%	34.043%	29.708%	27.660%
10	24.843%	34.043%	**29.737%**	**21.277%**
11	25.423%	34.043%	29.708%	23.404%

Table 28, Table 29, and Table 30 display the results of our experiments with model PCA-5 whose independent variables were selected in [46] for Releases 2, 3, and 4. Based on the results of the Release 2, the best results for the PCA-5 model were found to be for 9 similar cases, the weight of *USAGE* of 6 with a scale factor of 9.00. We compared these results to the results of the experiments with PCA-10 model.

Table 27: PCA-7 Model Release 4

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	27.226%	48.913%	32.347%	42.391%
2	28.230%	47.826%	33.608%	40.217%
3	28.513%	44.565%	33.634%	36.957%
4	28.487%	42.391%	34.174%	33.696%
5	28.616%	43.478%	34.225%	35.870%
6	28.821%	42.391%	34.457%	34.783%
7	28.744%	42.391%	34.714%	34.783%
8	28.693%	41.304%	34.689%	34.783%
9	28.873%	42.391%	34.225%	34.783%
10	28.976%	42.391%	**34.328%**	**32.609%**
11	29.104%	44.565%	34.508%	32.609%

Table 28: PCA-5 Model Release 2

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	27.453%	37.566%	32.015%	33.333%
2	28.059%	37.037%	32.806%	31.217%
3	27.980%	37.037%	32.911%	32.275%
4	27.505%	35.979%	32.358%	30.159%
5	25.791%	35.797%	32.094%	28.571%
6	26.820%	33.862%	**31.804%**	**28.571%**
7	26.767%	35.450%	31.593%	29.101%
8	26.714%	35.450%	31.145%	31.217%
10	26.292%	37.566%	31.013%	31.217%
11	26.319%	37.566%	30.670%	31.217%

Table 31, Table 32, and Table 33 display the results of our experiments with all 18 product and process PCA domain and execution metrics that were obtained in [46] for Releases 2, 3, and 4. Based on the model for the Release 2, the best results for the models with 18 variables were found with 9 similar cases and the weight of *USAGE* of 10, with a scale factor of 9.00.

Table 29: PCA-5 Model Release 3

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	29.937%	42.553%	34.717%	34.043%
2	29.794%	46.809%	34.946%	34.043%
3	29.622%	51.064%	34.659%	38.298%
4	29.508%	48.936%	34.402%	38.298%
5	29.393%	44.681%	34.373%	29.787%
6	29.479%	38.298%	**33.829%**	**29.787%**
7	29.193%	38.298%	33.429%	29.787%
8	28.993%	36.170%	33.200%	29.787%
10	29.021%	38.298%	33.314%	29.787%
11	28.935%	38.298%	33.457%	27.660%

Table 30: PCA-5 Model Release 4

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	33.402%	36.957%	39.629%	29.348%
2	34.431%	33.696%	41.122%	29.348%
3	35.023%	33.696%	41.019%	30.435%
4	35.383%	34.783%	41.173%	30.435%
5	35.615%	33.696%	41.431%	30.435%
6	35.306%	33.696%	**41.482%**	**28.261%**
7	35.718%	31.522%	41.457%	27.174%
8	35.769%	32.609%	41.457%	28.261%
10	35.950%	31.522%	41.740%	26.087%
11	35.692%	32.609%	41.765%	27.174%

Table 34, Table 35, and Table 36 display the results of our experiments with all 10 product PCA domain metrics that were obtained in [46] for Releases 2, 3, and 4. We had an easier time choosing the best model for the experiments with all 10 PCA product metrics. Based on Release 2, we were able to determine that the best model uses 9 similar cases and the *USAGE* weight at 1. This model was using a scale factor of 9.00.

Table 31: PCA-18 Model Release 2

weight	7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II
1	23.207%	43.386%	27.848%	38.624%
2	23.444%	43.386%	27.716%	37.566%
3	23.128%	42.328%	27.795%	37.037%
4	22.864%	43.915%	27.400%	37.566%
5	22.943%	43.386%	26.741%	37.566%
6	22.627%	42.857%	26.292%	37.566%
7	22.495%	42.328%	26.503%	38.095%
8	22.389%	42.328%	26.741%	37.037%
9	22.284%	41.799%	26.635%	37.037%
10	22.284%	42.857%	**26.714%**	**34.921%**
11	22.310%	42.857%	26.530%	35.979%

Table 32: PCA-18 Model Release 3

weight	7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II
1	23.354%	44.681%	27.962%	34.043%
2	24.041%	40.426%	28.420%	29.787%
3	24.299%	38.298%	28.392%	31.915%
4	24.442%	40.426%	28.649%	27.660%
5	23.927%	38.298%	28.477%	29.787%
6	23.612%	38.298%	28.563%	27.660%
7	23.497%	38.298%	27.991%	25.532%
8	23.412%	36.170%	27.876%	27.660%
9	23.068%	36.170%	27.647%	27.660%
10	22.896%	36.170%	**27.390%**	**27.660%**
11	22.982%	36.170%	27.161%	29.787%

C.3 Comparisons Between our Raw Metric Models And Our PCA-Generated Models

We experimented to see how the results of using CBR with the raw metrics compare to results achieved using CBR for the PCA-generated metrics. We had performed the

Table 33: PCA-18 Model Release 4

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	28.049%	36.957%	32.398%	32.609%
2	28.487%	34.783%	33.942%	30.435%
3	29.233%	36.957%	34.637%	30.435%
4	29.979%	35.870%	35.306%	30.435%
5	30.262%	34.783%	36.130%	29.348%
6	30.340%	35.870%	36.336%	29.348%
7	30.211%	34.783%	36.027%	30.435%
8	30.005%	34.783%	35.821%	30.435%
9	30.005%	35.870%	36.181%	29.348%
10	30.082%	38.043%	**36.078%**	**29.348%**
11	29.825%	39.130%	35.950%	29.348%

Table 34: PCA-10 Model Release 2

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	27.162%	36.508%	**32.120%**	**28.571%**
2	27.637%	35.979%	32.806%	30.688%
3	27.532%	35.450%	32.621%	30.688%
4	27.558%	37.037%	32.358%	31.217%
5	27.611%	37.566%	32.516%	32.275%
6	27.400%	38.095%	32.384%	33.333%
7	27.242%	39.153%	31.751%	32.804%
8	27.162%	38.624%	31.751%	31.746%
10	26.846%	40.212%	31.145%	33.862%
11	26.688%	39.683%	31.039%	33.862%

experiments for the 4 selected product metrics and for 11 selected product, process and execution as well as all 28 product metrics and compared the results of these models to the results of all four models that use PCA-generated metrics. During these experiments, we varied the weights for *USAGE*.

Our hypothesis was that the models using raw metrics will not be significantly worse

Table 35: PCA-10 Model Release 3

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	28.678%	38.298%	**33.944%**	**27.660%**
2	29.193%	31.915%	34.688%	21.277%
3	28.935%	31.915%	33.944%	21.277%
4	28.563%	31.915%	34.345%	21.277%
5	28.420%	34.043%	33.944%	21.277%
6	28.334%	34.043%	33.658%	19.149%
7	28.392%	34.043%	33.371%	14.894%
8	28.392%	27.660%	33.257%	14.894%
9	28.477%	27.660%	33.343%	14.894%
10	28.535%	25.532%	33.028%	14.894 %
11	28.248%	27.660%	32.799%	14.894%

Table 36: PCA-10 Model Release 4

weight of *USAGE*	7 cases Type I	7 cases Type II	9 cases Type I	9 cases Type II
1	32.141%	31.522%	**38.111%**	**27.174%**
2	33.711%	29.348%	39.887%	22.826%
3	34.303%	27.174%	40.093%	22.826%
4	34.380%	29.348%	40.170%	25.000%
5	34.766%	31.522%	40.607%	28.261%
6	34.817%	31.522%	41.045%	28.261%
7	34.792%	34.783%	40.736%	29.348%
8	34.586%	32.609%	40.968%	29.348%
9	34.792%	32.609%	41.045%	28.261%
10	34.689%	33.696%	40.736%	28.261%
11	34.740%	32.609%	40.839%	28.261%

than the models using PCA-generated metrics. If this statement is true, we can again show that CBR does not need preprocessing of any sort (such as PCA) to create an acceptable model. Table 37, Table 38, and Table 39, display the results of our experiments with all 28 raw product metrics for Releases 2, 3, and 4. These experiments were carried out to see whether it is necessary to pre-select significant raw metrics before using a CBR

Table 37: Raw-28 Model Release 2

weight	7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II
1	27.901%	37.037%	32.595%	31.217%
2	27.215%	36.508%	32.147%	31.217%
3	26.978%	36.508%	32.384%	33.333%
4	26.688%	39.153%	31.587%	33.862%
5	26.572%	40.741%	31.646%	33.862%
6	26.477%	39.153%	31.672%	33.862%
7	26.292%	37.566%	31.725%	33.333%
8	25.686%	39.153%	**30.459%**	**31.217%**
10	25.976%	38.095%	30.643%	32.275%

Table 38: Raw-28 Model Release 3

weight	7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II
1	29.136%	31.915%	33.600%	25.532%
2	29.136%	25.532%	33.515%	21.277%
3	28.706%	31.915%	33.371%	25.532%
4	28.506%	34.043%	33.171%	21.277%
5	28.592%	27.660%	33.400%	23.404%
6	28.477%	27.660%	33.457%	25.532%
7	28.392%	27.660%	33.257%	25.532%
8	28.392%	27.660%	**32.570%**	**25.532%**
10	28.077%	27.660%	28.077%	27.660%

tool. Here, we were able to see that the best results were achieved by selecting a model that uses 9 similar cases with the weight of *USAGE* at 2 (scale factor equals 9.00).

Table 39: Raw-28 Model Release 4

weight	7 cases		9 cases	
of *USAGE*	Type I	Type II	Type I	Type II
1	35.358%	36.957%	40.736%	28.261%
2	36.516%	33.696%	41.971%	23.913%
3	36.670%	35.870%	42.280%	27.174%
4	36.644%	33.696%	42.460%	25.000%
5	35.924%	28.261%	42.203%	23.913%
6	36.361%	29.348%	42.357%	22.826%
7	36.464%	29.348%	42.383%	23.913%
8	36.490%	29.348%	**42.383%**	**22.826%**
10	36.902%	28.261%	42.460%	21.739%